Workshop Réplication Logique

Réplication Logique : présentation et exercice pratique

Réplication Logique : présentation et exercice pratique

18.06

Dalibo & Contributors

http://dalibo.com/formations

Réplication Logique : présentation et exercice pratique

Workshop Réplication Logique

TITRE : Réplication Logique : présentation et exercice pratique
SOUS-TITRE : Workshop Réplication Logique

REVISION: 18.06

Table des Matières

1 RÉPLICATION LOGIQUE : PRÉSENTATION ET EXERCICE PRATIQUE

Figure 1: PostgreSQL

Photographie obtenue sur urltarget.com[1] .

Public Domain CC0.

[1] http://www.urltarget.com/art-rock-carving-elephant-sculpture-cyril.html

1.1 INTRODUCTION

- Principes
- Mise en place
  - entre 2 versions majeures différentes
- Supervision
- Limitations

La version 10 ajoute la réplication logique à PostgreSQL. Cette réplication était attendue depuis longtemps. Cet atelier permet de comprendre les principes derrière ce type de réplication, sa mise en place, son administration et sa supervision.

1.2 PRINCIPES

- Réplication physique
  - depuis la 9.0
  - beaucoup de possibilités
  - mais des limitations
- Réplication logique
  - permet de résoudre certaines des limitations de la réplication physique
  - auparavant uniquement disponible via des solutions externes
  - en interne depuis la version 10

La réplication existe dans PostgreSQL depuis la version 9.0. Il s'agit d'une réplication physique, autrement dit par application de bloc d'octets ou de delta de bloc. Ce type de réplication a beaucoup évolué au fil des versions 9.X mais a des limitations difficilement contournables directement.

La réplication logique apporte des réponses à ces limitations. Seules des solutions tierces apportaient ce type de réplication à PostgreSQL. Il a fallu attendre la version 10 pour la voir intégrer en natif.

1.2.1 RÉPLICATION PHYSIQUE VS LOGIQUE

- Réplication physique
  - instance complète
  - par bloc
  - asymétrique
  - asynchrone/synchrone
- Réplication logique
  - par table
  - par type d'opération
  - asymétrique (une origine des modifications)
  - asynchrone/synchrone

La réplication physique est une réplication au niveau bloc. Le serveur primaire envoie au secondaire les octets à ajouter/remplacer dans des fichiers. Le serveur secondaire n'a aucune information sur les objets logiques (tables, index, vues matérialisées, bases de données). Il n'y a donc pas de granularité possible, c'est forcément l'instance complète qui est répliquée. Cette réplication est par défaut en asynchrone mais il est possible de la configurer en synchrone suivant différents modes.

La réplication logique est une réplication du contenu des tables. Plus précisément, elle réplique les résultats des ordres SQL exécutés sur la table publiée et l'applique sur la table cible. La table cible peut être modifiée et son contenu différer de la table source.

Elle se paramètre donc table par table, et même opération par opération. Elle est asymétrique dans le sens où il existe une seule origine des écritures pour une table. Cependant, il est possible de réaliser des réplications croisées où un ensemble de tables est répliqué du serveur 1 vers le serveur 2 et un autre ensemble de tables est répliqué du serveur 2 vers le serveur 1. Enfin, elle fonctionne en asynchrone ou en synchrone.

1.2.2 LIMITATIONS DE LA RÉPLICATION PHYSIQUE

- Pas de réplication partielle
- Pas de réplication entre différentes versions majeures
- Pas de réplication entre différentes architectures
- Pas de réplication multidirectionnelle

Malgré ses nombreux avantages, la réplication physique souffre de quelques défauts.

Il est impossible de ne répliquer que certaines bases ou que certaines tables (pour ne pas répliquer des tables de travail par exemple). Il est aussi impossible de créer des in-

dex spécifiques ou même des tables de travail, y compris temporaires, sur les serveurs secondaires, vu qu'ils sont strictement en lecture seule.

Un serveur secondaire ne peut se connecter qu'à un serveur primaire de même version majeure. On ne peut donc pas se servir de la réplication physique pour mettre à jour la version majeure du serveur.

Enfin, il n'est pas possible de faire de la réplication entre des serveurs d'architectures matérielles ou logicielles différentes (32/64 bits, little/big endian, version de bibliothèque C, etc.).

La réplication logique propose une solution à tous ces problèmes, en dehors de la réplication multidirectionnelle.

1.2.3 QUELQUES TERMES ESSENTIELS

- Serveur origine
  - et serveurs de destination
- Publication
  - et abonnement

Dans le cadre de la réplication logique, on ne réplique pas une instance vers une autre. On publie les modifications effectuées sur le contenu d'une table à partir d'un serveur. Ce serveur est le serveur origine. De lui sont enregistrées les modifications que d'autres serveurs pourront récupérer. Ces serveurs de destination indiquent leur intérêt sur ces modifications en s'abonnant à la publication.

De ceci, il découle que :

- le serveur origine est le serveur où les écritures sur une table sont enregistrées pour publication vers d'autres serveurs ;
- les serveurs intéressés par ces enregistrements sont les serveurs destinations ;
- un serveur origine doit proposer une publication des modifications ;
- les serveurs destinations intéressés doivent s'abonner à une publication.

Dans un cluster de réplication, un serveur peut avoir un rôle de serveur origine ou de serveur destination. Il peut aussi avoir les deux rôles. Dans ce cas, il sera origine pour certaines tables et destinations pour d'autres. Il ne peut pas être à la fois origine et destination pour la même table.

1.2.4 RÉPLICATION EN FLUX

- Paramètre `wal_level`
- Processus `wal sender`
  - mais pas de `wal receiver`
  - un `logical replication worker` à la place
- Asynchrone / synchrone
- Slots de réplication

La réplication logique utilise le même canal d'informations que la réplication physique : les enregistrements des journaux de transactions. Pour que les journaux disposent de suffisamment d'informations, le paramètre `wal_level` doit être configuré en adéquation.

Une fois cette configuration effectuée et PostgreSQL redémarré sur le serveur origine, le serveur destination pourra se connecter au serveur origine dans le cadre de la réplication. Lorsque cette connexion est faite, un processus `wal sender` apparaîtra sur le serveur origine. Ce processus sera en communication avec un processus `logical replication worker` sur le serveur destination.

Comme la réplication physique, la réplication logique peut être configurée en asynchrone comme en synchrone, suivant le même paramétrage (`synchronous_commit`, `synchronous_standby_names`).

Chaque abonné maintient un slot de réplication sur l'instance de l'éditeur. Par défaut il est créé et supprimé automatiquement. La copie initiale des données crée également des slots de réplication temporaires.

1.2.5 GRANULARITÉ

- Par table
  - publication pour toutes les tables
  - publications pour des tables spécifiques
- Par opération
  - insert, update, delete

La granularité de la réplication physique est simple : c'est l'instance et rien d'autre.

Avec la réplication logique, la granularité est la table. Une publication se crée en indiquant la table pour laquelle on souhaite publier les modifications. On peut en indiquer plusieurs. On peut en ajouter après en modifiant la publication. Cependant, une nouvelle table ne sera pas ajoutée automatiquement à la publication. Ceci n'est possible que dans un cas

précis : la publication a été créée en demandant la publication de toutes les tables (clause `FOR ALL TABLES`).

La granularité peut aussi se voir au niveau des opérations de modification réalisées. On peut très bien ne publier que les opérations d'insertion, de modification ou de suppression. Par défaut, tout est publié.

1.2.6 LIMITATIONS DE LA RÉPLICATION LOGIQUE

- Pas de réplication des requêtes DDL
  - et donc pas de `TRUNCATE`
- Pas de réplication des valeurs des séquences
- Pas de réplication des LO (table système)
- Contraintes d'unicité obligatoires pour les `UPDATE`/`DELETE`
- Coût en CPU et I/O

La réplication logique n'a pas que des atouts, elle a aussi ses propres limitations.

La première, et plus importante, est qu'elle ne réplique que les changements de données des tables. Donc une table nouvellement créée ne sera pas forcément répliquée. L'ajout (ou la suppression) d'une colonne ne sera pas répliqué, causant de ce fait un problème de réplication quand l'utilisateur y ajoutera des données.

Il n'y a pas non plus de réplication des valeurs des séquences. Les valeurs des séquences sur les serveurs destinations seront donc obsolètes.

Les `Large Objects` étant stockés dans une table système, ils ne sont pas pris en compte par la réplication logique.

Les opérations `UPDATE` et `DELETE` nécessitent la présence d'une contrainte unique pour s'assurer de modifier ou supprimer les bonnes lignes.

Enfin, la réplication logique a un coût en CPU (sur les deux instances concernées) comme en écritures disques relativement important : attention aux petites configurations.

1.3 MISE EN PLACE

- Cas simple
  - 2 serveurs
  - une seule origine
  - un seul destinataire
  - une seule publication
- Plusieurs étapes
  - configuration du serveur origine
  - configuration du serveur destination
  - création d'une publication
  - ajout d'une souscription

Dans cette partie, nous allons aborder un cas simple avec uniquement deux serveurs. Le premier sera l'origine, le second sera le destinataire des informations de réplication. Toujours pour simplifier l'explication, il n'y aura pour l'instant qu'une seule publication.

La mise en place de la réplication logique consiste en plusieurs étapes :

- la configuration du serveur origine ;
- la configuration du serveur destination ;
- la création d'une publication ;
- l'abonnement à une publication.

Nous allons voir maintenant ces différents points.

1.3.1 CONFIGURER LE SERVEUR ORIGINE

- Création et configuration de l'utilisateur de réplication
  - et lui donner les droits de lecture des tables à répliquer
- Configuration du fichier `postgresql.conf`
  - `wal_level = logical`
- Configuration du fichier `pg_hba.conf`
  - autoriser une connexion de réplication du serveur destination

Dans le cadre de la réplication avec PostgreSQL, c'est toujours le serveur destination qui se connecte au serveur origine. Pour la réplication physique, on utilise plutôt les termes de serveur primaire et de serveur secondaire mais c'est toujours du secondaire vers le primaire, de l'abonné vers le publieur.

Tout comme pour la réplication physique, il est nécessaire de disposer d'un utilisateur PostgreSQL capable de se connecter au serveur origine et capable d'initier une connexion de réplication. Voici donc la requête pour créer ce rôle :

```
CREATE ROLE logrepli LOGIN REPLICATION;
```

Cet utilisateur doit pouvoir lire le contenu des tables répliquées. Il lui faut donc le droit `SELECT` sur ces objets :

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO logrepli;
```

Les journaux de transactions doivent disposer de suffisamment d'informations pour que le `wal sender` puisse envoyer les bonnes informations au `logical replication worker`. Pour cela, il faut configurer le paramètre `wal_level` à la valeur `logical` dans le fichier `postgresql.conf`.

Enfin, la connexion du serveur destination doit être possible sur le serveur origine. Il est donc nécessaire d'avoir une ligne du style :

```
host base_publication logrepli XXX.XXX.XXX.XXX/XX md5
```

en remplaçant `XXX.XXX.XXX.XXX/XX` par l'adresse CIDR du serveur destination. La méthode d'authentification peut aussi être changée suivant la politique interne. Suivant la méthode d'authentification, il sera nécessaire ou pas de configurer un mot de passe pour cet utilisateur.

Si le paramètre `wal_level` a été modifié, il est nécessaire de redémarrer le serveur PostgreSQL. Si seul le fichier `pg_hba.conf` a été modifié, seul un rechargement de la configuration est demandé.

1.3.2 TP : CONFIGURATION DU SERVEUR ORIGINE S1

- Création et configuration de l'utilisateur de réplication

```
CREATE ROLE logrepli LOGIN REPLICATION;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO logrepli;
```

- Fichier `postgresql.conf`

```
wal_level = logical
```

- Fichier `pg_hba.conf`

```
local b1 logrepli trust
```

- Redémarrer le serveur origine
- Attention, dans la vraie vie, ne pas utiliser `trust`
  - et utiliser le fichier `.pgpass`

La configuration du serveur d'origine commence par la création du rôle de réplication. On lui donne ensuite les droits sur toutes les tables. Ici, la commande ne s'occupe que des tables du schéma `public`, étant donné que nous n'avons que ce schéma. Dans le cas où la base dispose d'autres schémas, il serait nécessaire d'ajouter les ordres SQL pour ces schémas.

Les fichiers `postgresql.conf` et `pg_hba.conf` sont modifiés pour y ajouter la configuration nécessaire. Le serveur PostgreSQL du serveur d'origine est alors redémarré pour qu'il prenne en compte cette nouvelle configuration.

Il est important de répéter que la méthode d'authentification `trust` ne devrait jamais être utilisée en production. Elle n'est utilisée ici que pour se faciliter la vie.

1.3.3 CONFIGURER LE SERVEUR DESTINATION

- Création de l'utilisateur de réplication
- Création, si nécessaire, des tables répliquées
  - `pg_dump -h serveur_origine -s -t la_table la_base | psql la_base`

Sur le serveur destination, il n'y a pas de configuration à réaliser dans les fichiers `postgresql.conf` et `pg_hba.conf`.

Cependant, il est nécessaire d'avoir l'utilisateur de réplication. La requête de création est identique :

```
CREATE ROLE logrepli LOGIN REPLICATION;
```

Ensuite, il faut récupérer la définition des objets répliqués pour les créer sur le serveur de destination. Le plus simple est d'utiliser `pg_dump` pour cela et d'envoyer le résultat directement à `psql` pour restaurer immédiatement les objets. Cela se fait ainsi :

```
pg_dump -h serveur_origine --schema-only base | psql base
```

Il est possible de sauvegarder la définition d'une seule table en ajoutant l'option `-t` suivi du nom de la table.

1.3.4 TP : CONFIGURER LE SERVEUR DESTINATION S2

- Création de l'utilisateur de réplication

```
CREATE ROLE logrepli LOGIN REPLICATION;
```

- Création des tables répliquées (sans contenu)

```
createdb -p 5433 b1
pg_dump -p 5432 -s b1 | psql -p 5433 b1
```

La configuration consiste en la création de l'utilisateur de réplication. Puis, nous utilisons `pg_dump` pour récupérer la définition de tous les objets grâce à l'option `-s` (ou `--schema-only`). Ces ordres SQL sont passés à `psql` pour qu'il les intègre dans la base b1 du serveur destination.

1.3.5 CRÉER UNE PUBLICATION COMPLÈTE

- Ordre SQL

```
CREATE PUBLICATION nom
  [ FOR TABLE [ ONLY ] nom_table [ * ] [, ...]
    | FOR ALL TABLES ]
  [ WITH ( parametre_publication [= valeur] [, ... ] ) ]
```

- parametre_publication étant seulement le paramètre publish
  - valeurs possibles : insert, update, delete
  - les trois par défaut

Une fois que les tables sont définies des deux côtés (origine et destination), il faut créer une publication sur le serveur origine. Cette publication indiquera à PostgreSQL les tables répliquées et les opérations concernées.

La clause `FOR ALL TABLES` permet de répliquer toutes les tables de la base, sans avoir à les nommer spécifiquement. De plus, toute nouvelle table sera répliquée automatiquement dès sa création.

Si on ne souhaite répliquer qu'un sous-ensemble, il faut dans ce cas spécifier toutes les tables à répliquer en utilisant la clause `FOR TABLE` et en séparant les noms des tables par des virgules.

Cette publication est concernée par défaut par toutes les opérations d'écriture (`INSERT`, `UPDATE`, `DELETE`). Cependant, il est possible de préciser les opérations si on ne les souhaite pas toutes. Pour cela, il faut utiliser le paramètre de publication `publish` en utilisant les valeurs `insert`, `update` et/ou `delete` et en les séparant par des virgules si on en indique plusieurs.

1.3.6 TP : CRÉER UNE PUBLICATION COMPLÈTE

- Création d'une publication de toutes les tables de la base b1 sur le serveur origine s1

```
CREATE PUBLICATION publi_complete
  FOR ALL TABLES;
```

On utilise la clause `ALL TABLES` pour une réplication complète d'une base.

1.3.7 SOUSCRIRE À UNE PUBLICATION

- Ordre SQL

```
CREATE SUBSCRIPTION nom
    CONNECTION 'infos_connexion'
    PUBLICATION nom_publication [, ...]
    [ WITH ( parametre_souscription [= value] [, ... ] ) ]
```

- infos_connexion est la chaîne de connexion habituelle

Une fois la publication créée, le serveur destination doit s'y abonner. Il doit pour cela indiquer sur quel serveur se connecter et à quelle publication souscrire.

Le serveur s'indique avec la chaîne `infos_connexion`, dont la syntaxe est la syntaxe habituelle des chaînes de connexion. Pour rappel, on utilise les mots clés `host`, `port`, `user`, `password`, `dbname`, etc.

Le champ `nom_publication` doit être remplacé par le nom de la publication créé précédemment sur le serveur origine.

Les paramètres de souscription sont détaillés dans la slide suivante.

1.3.8 OPTIONS DE LA SOUSCRIPTION

- `copy_data`
  - copie initiale des données (activé par défaut)
- `create_slot`
  - création du slot de réplication (activé par défaut)
- `enabled`
  - activation immédiate de la souscription (activé par défaut)
- `slot_name`
  - nom du slot (par défaut, le nom de la souscription)
- `synchronous_commit`
  - pour surcharger la valeur du paramètre `synchronous_commit`
- `connect`
  - connexion immédiate (activé par défaut)

Les options de souscription sont assez nombreuses et permettent de créer une souscription pour des cas particuliers. Par exemple, si le serveur destination a déjà les données du serveur origine, il faut placer le paramètre `copy_data` à la valeur `off`.

1.3.9 TP : SOUSCRIRE À LA PUBLICATION

- Souscrire sur s2 à la publication de s1

```
CREATE SUBSCRIPTION subscr_complete
  CONNECTION 'port=5432 user=logrepli dbname=b1'
  PUBLICATION publi_complete;
```

- Un slot de réplication est créé
- Les données initiales sont immédiatement transférées

Maintenant que le serveur s1 est capable de publier les informations de réplication, le serveur intéressé doit s'y abonner. Lors de la création de la souscription, il doit préciser comment se connecter au serveur origine et le nom de la publication.

La création de la souscription ajoute immédiatement un slot de réplication sur le serveur origine.

Les données initiales de la table t1 sont envoyées du serveur s1 vers le serveur s2.

1.3.10 TP : TESTS DE LA RÉPLICATION COMPLÈTE

- Insertion, modification, suppression sur les différentes tables de s1
- Vérifications sur s2
  - toutes doivent avoir les mêmes données entre s1 et s2

Toute opération d'écriture sur la table t1 du serveur s1 doit être répliquée sur le serveur s2.

Sur le serveur s1 :

```
b1=# INSERT INTO t1 VALUES (101, 't1, ligne 101');
INSERT 0 1
b1=# UPDATE t1 SET label_t1=upper(label_t1) WHERE id_t1=10;
UPDATE 1
b1=# DELETE FROM t1 WHERE id_t1=11;
DELETE 1
b1=# SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
 id_t1 |   label_t1
-------+---------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
(2 rows)
```

Sur le serveur s2 :

```
b1=# SELECT count(*) FROM t1;
 count
-------
   100
(1 row)

b1=# SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
 id_t1 |   label_t1
-------+---------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
(2 rows)
```

1.3.11 RÉPLICATION PARTIELLE

- Identique à la réplication complète, à une exception...
- Créer la publication partielle

```
CREATE PUBLICATION publi_partielle
  FOR TABLE t1,t2;
```

La mise en place d'une réplication partielle est identique à la mise en place d'une réplication complète à une exception. La publication doit mentionner la liste des tables à répliquer. Chaque nom de table est séparé par une virgule.

Cela donne donc dans notre exemple :

```
CREATE PUBLICATION publi_partielle
  FOR TABLE t1,t2;
```

1.3.12 RÉPLICATION CROISÉE

- On veut pouvoir écrire sur une table sur le serveur s1
  - et répliquer les écritures de cette table sur s2
- On veut aussi pouvoir écrire sur une (autre) table sur le serveur s2
  - et répliquer les écritures de cette table sur s1

La réplication logique ne permet pas pour l'instant de faire du multidirectionnel (multi-maître) pour une même table. Cependant, il est tout à fait possible de faire en sorte qu'un ensemble de tables soit répliqué du serveur s1 (origine) vers le serveur s2 et qu'un autre ensemble de tables soit répliqué du serveur s2 (origine) vers le serveur s1 (destination).

1.4 SUPERVISION

- Méta-données
- Statistiques

1.5 CATALOGUES SYSTÈMES - MÉTA-DONNÉES

- `pg_publication`
  - définition des publications
  - `\dRp` sous psql
- `pg_publication_tables`
  - tables ciblées par chaque publication
- `pg_subscription`
  - définition des souscriptions
  - `\dRs` sous psql

Le catalogue système `pg_publication` contient la liste des publications, avec leur méta-données :

```
b1=# SELECT * FROM pg_publication;
     pubname     | pubowner | puballtables | pubinsert | pubupdate | pubdelete
-----------------+----------+--------------+-----------+-----------+-----------
 publi_complete  |       10 | t            | t         | t         | t
(1 row)
```

Le catalogue système `pg_publication_tables` contient une ligne par table par publication :

```
b1=# SELECT * FROM pg_publication_tables;
    pubname     | schemaname | tablename
----------------+------------+-----------
 publi_complete | public     | t1
 publi_complete | public     | t2
(2 rows)
```

On peut en déduire deux versions abrégées :

- la liste des tables par publication :

```
SELECT pubname, array_agg(tablename ORDER BY tablename) AS tables_list
FROM pg_publication_tables
GROUP BY 1
ORDER BY 1;

    pubname     | tables_list
----------------+-------------
 publi_complete | {t1,t2}
(1 row)
```

- la liste des publications par table :

```
SELECT tablename, array_agg(pubname ORDER BY pubname) AS publications_list
FROM pg_publication_tables
GROUP BY 1
ORDER BY 1;

 tablename | publications_list
-----------+-------------------
 t1        | {publi_complete}
 t2        | {publi_complete}
(2 rows)
```

Enfin, il y a aussi un catalogue système contenant la liste des souscriptions :

```
b1=# \x
Expanded display is on.
b1=# SELECT * FROM pg_subscription;
-[ RECORD 1 ]---+---------------------------------
subdbid         | 16384
subname         | subscr_complete
subowner        | 10
subenabled      | t
subconninfo     | port=5432 user=logrepli dbname=b1
subslotname     | subscr_complete
subsynccommit   | off
subpublications | {publi_complete}
```

1.6 VUES STATISTIQUES

- `pg_stat_replication`
  - statut de réplication
- `pg_stat_subscription`
  - état des souscriptions
- `pg_replication_origin_status`
  - statut des origines de réplication

Comme pour la réplication physique, le retard de réplication est calculable en utilisant les informations de la vue `pg_stat_replication` :

```
b1=# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 19854
usesysid         | 16407
usename          | logrepli
application_name | subscr_complete
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-08-27 14:55:27.85201+02
backend_xmin     |
state            | streaming
sent_lsn         | 0/16E1F68
write_lsn        | 0/16E1F68
flush_lsn        | 0/16E1F68
replay_lsn       | 0/16E1F68
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
```

L'état des souscriptions est disponible sur les serveurs destination à partir de la vue `pg_stat_subscription` :

```
b1=# SELECT * FROM pg_stat_subscription;
-[ RECORD 1 ]---------+------------------------------
subid                 | 16408
subname               | subscr_complete
pid                   | 19853
relid                 |
received_lsn          | 0/16E1F68
last_msg_send_time    | 2018-08-27 14:56:04.685138+02
last_msg_receipt_time | 2018-08-27 14:56:04.68524+02
latest_end_lsn        | 0/16E1F68
latest_end_time       | 2018-08-27 14:56:04.685138+02
```

1.7 POSSIBILITÉS SUR LES TABLES RÉPLIQUÉES

- Possibilités sur les tables répliquées :
  - Index supplémentaires
  - Modification des valeurs
  - Colonnes supplémentaires
  - Triggers également activables sur la table répliquée
- Attention à la cohérence

La réplication logique permet plusieurs choses impensables en réplication physique. Les cas d'utilisation sont en fait très différents.

On peut rajouter ou supprimer des index sur la table répliquée, pourvu que les lignes restent identifiables. Au besoin on peut préciser l'index, qui doit être unique sur colonne `NOT NULL` servant de clé :

```
ALTER TABLE nomtable REPLICA IDENTITY USING INDEX nomtable_col_idx;
```

Il est possible de modifier des valeurs dans la table répliquée. Ces modifications sont susceptibles d'être écrasées par des modifications de la table source sur les mêmes lignes. Il est aussi possible de perdre la synchronisation entre les tables, notamment si on modifie la clé primaire.

Les triggers ne se déclenchent par défaut que sur la base d'origine. On peut activer ainsi un trigger sur la table répliquée :

```
ALTER TABLE matable ENABLE REPLICA TRIGGER nom_trigger ;
```

Tout cela est parfois très pratique mais peut poser de sérieux problème de cohérence de données entre les deux instances si l'on ne fait pas attention. On vérifiera régulièrement les erreurs dans les traces.

1.7.1 EMPÊCHER LES ÉCRITURES SUR UN SERVEUR DESTINATION

- Par défaut, toutes les écritures sont autorisées sur le serveur destination
  - y compris écrire dans une table répliquée avec un autre serveur comme origine
- Problème
  - serveurs non synchronisés
  - blocage de la réplication en cas de conflit sur la clé primaire
- Solution
  - révoquer le droit d'écriture sur le serveur destination

– mais ne pas révoquer ce droit pour le rôle de réplication !

Sur s2, nous allons créer un utilisateur applicatif en lui donnant tous les droits :

```
b1=# CREATE ROLE u1 LOGIN;
CREATE ROLE
b1=# GRANT ALL ON ALL TABLES IN SCHEMA public TO u1;
GRANT
```

L'autoriser à se connecter dans le pg_hba.conf en y ajoutant :

```
local b1 u1 trust
```

Recharger la configuration. Maintenant, nous nous connectons avec cet utilisateur et vérifions s'il peut écrire dans la table répliquée :

```
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> INSERT INTO t1 VALUES (103, 't1 sur s2, ligne 103');
INSERT 0 1
```

C'est bien le cas, contrairement à ce que l'on aurait pu croire instinctivement. Le seul moyen d'empêcher ce comportement par défaut est de lui supprimer les droits d'écriture :

```
b1=> \c b1 postgres
You are now connected to database "b1" as user "postgres".
b1=# REVOKE INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public FROM u1;
REVOKE
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> INSERT INTO t1 VALUES (104);
ERROR:  permission denied for relation t1
```

L'utilisateur u1 ne peut plus écrire dans les tables répliquées.

Si cette interdiction n'est pas réalisée, on peut arriver à des problèmes très gênants. Par exemple, nous avons inséré dans la table `t1` de s2 la valeur 103 :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s2, ligne 103
(1 row)
```

Cette ligne n'apparaît pas sur s1 :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 | label_t1
-------+----------
(0 rows)
```

De ce fait, on peut l'insérer sur la table t1 de s1 :

```
b1=> INSERT INTO t1 VALUES (103, 't1 sur s1, ligne 103');
INSERT 0 1
```

Et maintenant, on se trouve avec deux serveurs désynchronisés :

- sur s1 :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s1, ligne 103
(1 row)
```

- sur s2 :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s2, ligne 103
(1 row)
```

Notez que le contenu de la colonne `label_t1` n'est pas identique sur les deux serveurs.

Le processus de réplication logique n'arrive plus à appliquer les données sur s2, d'où les messages suivants dans les traces :

```
LOG:  logical replication apply worker for subscription "subscr_complete"
has started
ERROR:  duplicate key value violates unique constraint "t1_pkey"
DETAIL:  Key (id_t1)=(103) already exists.
LOG:  background worker "logical replication worker" (PID 19923) exited
with exit code 1
```

Il faut corriger manuellement la situation, par exemple en supprimant la ligne de `t1` sur le serveur s2 :

```
b1=# DELETE FROM t1 WHERE id_t1=103;
DELETE 1
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 | label_t1
```

```
-------+----------
(0 rows)
```

Au bout d'un certain temps, le worker est relancé, et la nouvelle ligne est finalement disponible :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s1, ligne 103
(1 row)
```

1.7.2 QUE FAIRE POUR LES DDL

- Les opérations DDL ne sont pas répliquées
- De nouveaux objets ?
  - les déclarer sur tous les serveurs du cluster de réplication
  - tout du moins, ceux intéressés par ces objets
- Changement de définition des objets ?
  - à réaliser sur chaque serveur

Seules les opérations DML sont répliquées pour les tables ciblées par une publication.

Toutes les opérations DDL sont ignorées, que ce soit l'ajout, la modification ou la suppression d'un objet, y compris si cet objet fait partie d'une publication.

Il est donc important que toute modification de schéma soit effectuée sur toutes les instances d'un cluster de réplication. Ce n'est cependant pas requis. Il est tout à fait possible d'ajouter un index sur un serveur sans vouloir l'ajouter sur d'autres. C'est d'ailleurs une des raisons de passer à la réplication logique.

Par contre, dans le cas du changement de définition d'une table répliquée (ajout ou suppression d'une colonne, par exemple), il est nettement préférable de réaliser cette opération sur tous les serveurs intégrés dans cette réplication.

1.7.3 QUE FAIRE POUR LES NOUVELLES TABLES

- Publication complète
  - rafraîchir les souscriptions concernées
- Publication partielle
  - ajouter la nouvelle table dans les souscriptions concernées

La création d'une table est une opération DDL. Elle est donc ignorée dans le contexte de la réplication logique. Il est tout à fait concevable qu'on ne veuille pas la répliquer, auquel cas il n'y a rien besoin de faire. Mais si on souhaite répliquer son contenu, deux cas se présentent : la publication a été déclarée `FOR ALL TABLES` ou elle a été déclarée pour certaines tables uniquement.

Dans le cas où la publication ne concerne qu'un sous-ensemble de tables, il faut ajouter la table à la publication avec l'ordre `ALTER PUBLICATION...ADD TABLE`.

Dans le cas où elle a été créé avec la clause `FOR ALL TABLES`, la nouvelle table est immédiatement prise en compte dans la publication. Cependant, pour que les serveurs destinataires gèrent aussi cette nouvelle table, il va falloir leur demander de rafraîchir leur souscription avec l'ordre `ALTER SUBSCRIPTION...REFRESH PUBLICATION`.

Voici un exemple de ce deuxième cas.

Sur le serveur s1, on crée la table `t4`, on lui donne les bons droits, et on insère des données :

```
b1=# CREATE TABLE t4 (id_t4 integer, primary key (id_t4));
CREATE TABLE
b1=# GRANT SELECT ON TABLE t4 TO logrepli;
GRANT
b1=# INSERT INTO t4 VALUES (1);
INSERT 0 1
```

Sur le serveur s2, on regarde le contenu de la table `t4` :

```
b1=# SELECT * FROM t4;
ERROR:  relation "t4" does not exist
LINE 1: SELECT * FROM t4;
                      ^
```

La table n'existe pas. En effet, la réplication logique ne s'occupe que des modifications de contenu des tables, pas des changements de définition. Il est donc nécessaire de créer la table sur le serveur destination, ici s2 :

```
b1=# CREATE TABLE t4 (id_t4 integer, primary key (id_t4));
```

```
CREATE TABLE
b1=# SELECT * FROM t4;
 id_t4
-------
(0 rows)
```

Elle ne contient toujours rien. Ceci est dû au fait que la souscription n'a pas connaissance de la réplication de cette nouvelle table. Il faut donc rafraîchir les informations de souscription :

```
b1=# ALTER SUBSCRIPTION subscr_complete REFRESH PUBLICATION;
ALTER SUBSCRIPTION
b1=# SELECT * FROM t4;
 id_t4
-------
     1
(1 row)
```

1.8 RAPPEL DES LIMITATIONS

- Pas de réplication des requêtes DDL
  - et donc pas de `TRUNCATE`
- Pas de réplication des valeurs des séquences
- Pas de réplication des LO (table système)
- Contraintes d'unicité obligatoires pour les `UPDATE`/`DELETE`

1.9 CONCLUSION

- Enfin une réplication logique
- Réplication complète ou partielle
  - par objet (table)
  - par opération (insert/update/delete)

1.9.1 QUESTIONS

N'hésitez pas, c'est le moment !

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegardes et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **DEVSQLPG : SQL pour PostgreSQL**
  https://dali.bo/devsqlpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancé**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle vers PostgreSQL**
  https://dali.bo/migorpg

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.